CHAPTER 3 Algorithms

SECTION 3.1 Algorithms

2. **a)** This procedure is not finite, since execution of the **while** loop continues forever.

b) This procedure is not effective, because the step $m := 1/n$ cannot be performed when $n = 0$, which will eventually be the case.

c) This procedure lacks definiteness, since the value of i is never set.

d) This procedure lacks definiteness, since the statement does not tell whether x is to be set equal to a or to b.

4. Set the answer to be $-\infty$. For i going from 1 through $n-1$, compute the value of the $(i+1)^{\text{st}}$ element in the list minus the i^{th} element in the list. If this is larger than the answer, reset the answer to be this value.

6. We need to go through the list and count the negative entries.

```
procedure negatives($a_1, a_2, \ldots, a_n$ : integers)
$k := 0$
for $i := 1$ to $n$
        if $a_i < 0$ then $k := k + 1$
return $k$ {the number of negative integers in the list}
```

8. This is similar to Exercise 7, modified to keep track of the largest even integer we encounter.

```
procedure largest even location($a_1, a_2, \ldots, a_n$ : integers)
$k := 0$
$largest := -\infty$
for $i := 1$ to $n$
        if ($a_i$ is even and $a_i > largest$) then
                $k := i$
                $largest := a_i$
return $k$ {the desired location (or 0 if there are no evens)}
```

10. We assume that if the input $x = 0$, then $n > 0$, since otherwise x^n is not defined. In our procedure, we let $m = |n|$ and compute x^m in the obvious way. Then if n is negative, we replace the answer by its reciprocal.

```
procedure power($x$ : real number, $n$ : integer)
$m := |n|$
$power := 1$
for $i := 1$ to $m$
        $power := power \cdot x$
if $n < 0$ then $power := 1/power$
return $power$ {$power = x^n$}
```

12. Four assignment statements are needed, one for each of the variables and a temporary assignment to get started so that we do not lose one of the original values.

```
    $temp := x$
    $x := y$
    $y := z$
    $z := temp$
```

14. **a)** With linear search we start at the beginning of the list, and compare 7 successively with 1, 3, 4, 5, 6, 8, 9, and 11. When we come to the end of the list and still have not found 7, we conclude that it is not in the list.

b) We begin the search on the entire list, with $i = 1$ and $j = n = 8$. We set $m := 4$ and compare 7 to the fourth element of the list. Since $7 > 5$, we next restrict the search to the second half of the list, with $i = 5$ and $j = 8$. This time we set $m := 6$ and compare 7 to the sixth element of the list. Since $7 \not> 8$, we next restrict ourselves to the first half of the second half of the list, with $i = 5$ and $j = 6$. This time we set $m := 5$, and compare 7 to the fifth element. Since $7 > 6$, we now restrict ourselves to the portion of the list between $i = 6$ and $j = 6$. Since at this point $i \not< j$, we exit the loop. Since the sixth element of the list is not equal to 7, we conclude that 7 is not in the list.

16. We let *min* be the smallest element found so far. At the end, it is the smallest element, since we update it as necessary as we scan through the list.

```
procedure smallest($a_1, a_2, \ldots, a_n$ : natural numbers)
$min := a_1$
for $i := 2$ to $n$
    if $a_i < min$ then $min := a_i$
return $min$ {the smallest integer among the input}
```

18. This is similar to Exercise 17.

```
procedure last smallest($a_1, a_2, \ldots, a_n$ : integers)
$min := a_1$
$location := 1$
for $i := 2$ to $n$
    if $min \geq a_i$ then
        $min := a_i$
        $location := i$
return $location$ {the location of the last occurrence of the smallest element in the list}
```

20. We just combine procedures for finding the largest and smallest elements.

```
procedure smallest and largest($a_1, a_2, \ldots, a_n$ : integers)
$min := a_1$
$max := a_1$
for $i := 2$ to $n$
    if $a_i < min$ then $min := a_i$
    if $a_i > max$ then $max := a_i$
{ $min$ is the smallest integer among the input, and $max$ is the largest}
```

22. We assume that the input is a sequence of symbols, a_1, a_2, ..., a_n, each of which is either a letter or a blank. We build up the longest word in *word*; its length is *length*. We denote the empty word by λ.

```
procedure longest word($a_1, a_2, \ldots, a_n$ : symbols)
maxlength := 0
maxword := $\lambda$
i := 1
while $i \le n$
        word := $\lambda$
        length := 0
        while $a_i \ne$ blank and $i \le n$
                length := length + 1
                word := concatenation of word and $a_i$
                i := i + 1
        if length > max then
                maxlength := length
                maxword := word
        i := i + 1
return maxword {the longest word in the sentence}
```

24. This is similar to Exercise 23. We let the array *hit* keep track of which elements of the codomain B have already been found to be images of elements of the domain A. When we find an element that has already been hit being hit again, we conclude that the function is not one-to-one.

```
procedure one_one(f : function, $a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_m$ : integers)
for i := 1 to m
        $hit(b_i)$ := 0
one_one := true
for j := 1 to n
        if $hit(f(a_j)) = 0$ then $hit(f(a_j))$ := 1
        else one_one := false
return one_one
```

26. There are two changes. First, we need to test $x = a_m$ (right after the computation of m) and take appropriate action if equality holds (what we do is set i and j both to be m). Second, if $x \not> a_m$, then instead of setting j equal to m, we can set j equal to $m-1$. The advantages are that this allows the size of the "half" of the list being looked at to shrink slightly faster, and it allows us to stop essentially as soon as we have found the element we are looking for.

28. This could be thought of as just doing two iterations of binary search at once. We compare the sought-after element to the middle element in the still-active portion of the list, and then to the middle element of either the top half or the bottom half. This will restrict the subsequent search to one of four sublists, each about one-quarter the size of the previous list. We need to stop when the list has length three or less and make explicit checks. Here is the pseudocode.

```
procedure tetrary search($x$ : integer, $a_1, a_2, \ldots, a_n$ : increasing integers)
$i := 1$
$j := n$
while $i < j - 2$
    $l := \lfloor (i+j)/4 \rfloor$
    $m := \lfloor (i+j)/2 \rfloor$
    $u := \lfloor 3(i+j)/4 \rfloor$
    if $x > a_m$ then if $x \le a_u$ then
                $i := m + 1$
                $j := u$
          else $i := u + 1$
    else if $x > a_l$ then
                $i := l + 1$
                $j := m$
          else $j := l$
if $x = a_i$ then location := $i$
else if $x = a_j$ then location := $j$
else if $x = a_{\lfloor (i+j)/2 \rfloor}$ then location := $\lfloor (i+j)/2 \rfloor$
else location := 0
return location {the subscript of the term equal to $x$ (0 if not found)}
```

30. The following algorithm will find all modes in the sequence and put them into a list L. At each point in the execution of this algorithm, *modecount* is the number of occurrences of the elements found to occur most often so far (the elements in L). Whenever a more frequently occurring element is found (the main inner loop), *modecount* and L are updated; whenever an element is found with this same count, it is added to L.

```
procedure find all modes($a_1, a_2, \ldots, a_n$ : nondecreasing integers)
modecount := 0
$i := 1$
while $i \le n$
    value := $a_i$
    count := 1
    while $i \le n$ and $a_i$ = value
        count := count + 1
        $i := i + 1$
    if count > modecount then
        modecount := count
        set $L$ to consist just of value
    else if count = modecount then add value to $L$
return $L$ {the list of all the values occurring most often, namely modecount times}
```

32. The following algorithm will find all terms of a finite sequence of integers that are greater than the sum of all the previous terms. We put them into a list L, but one could just as easily have them printed out, if that were desired. It might be more useful to put the *indices* of these terms into L, rather than the terms themselves (i.e., their values), but we take the former approach for variety. As usual, the empty list is considered to have sum 0, so the first term in the sequence is included in L if and only if it positive.

```
procedure find all biggies($a_1, a_2, \ldots, a_n$ : integers)
set $L$ to be the empty list
sum := 0
$i := 1$
while $i \le n$
    if $a_i$ > sum then append $a_i$ to $L$
    sum := sum + $a_i$
    $i := i + 1$
return $L$ {the list of all the values that exceed the sum of all the previous terms in the sequence}
```

34. There are five passes through the list. After one pass the list reads $2, 3, 1, 5, 4, 6$, since the 6 is compared and moved at each stage. During the next pass, the 2 and the 3 are not interchanged, but the 3 and the 1 are, as are the 5 and the 4, yielding $2, 1, 3, 4, 5, 6$. On the third pass, the 2 and the 1 are interchanged, yielding $1, 2, 3, 4, 5, 6$. There are two more passes, but no further interchanges are made, since the list is now in order.

36. The procedure is the same as that given in the solution to Exercise 35. We will exhibit the lists obtained after each step, with all the lists obtained during one pass on the same line.

$dfkmab$, $dfkmab$, $dfkmab$, $dfkamb$, $dfkabm$
$dfkabm$, $dfkabm$, $dfakbm$, $dfabkm$
$dfabkm$, $dafbkm$, $dabfkm$
$adbfkm$, $abdfkm$
$abdfkm$

38. We start with $6, 2, 3, 1, 5, 4$. The first step inserts 2 correctly into the sorted list 6, producing $2, 6, 3, 1, 5, 4$. Next 3 is inserted into $2, 6$, and the list reads $2, 3, 6, 1, 5, 4$. Next 1 is inserted into $2, 3, 6$, and the list reads $1, 2, 3, 6, 5, 4$. Next 5 is inserted into $1, 2, 3, 6$, and the list reads $1, 2, 3, 5, 6, 4$. Finally 4 is inserted into $1, 2, 3, 5, 6$, and the list reads $1, 2, 3, 4, 5, 6$. At each insertion, the element to be inserted is compared with the elements already sorted, starting from the beginning, until its correct spot is found, and then the previously sorted elements beyond that spot are each moved one position toward the back of the list.

40. We start with d, f, k, m, a, b. The first step inserts f correctly into the sorted list d, producing no change. Similarly, no change results when k and m are inserted into the sorted lists d, f and d, f, k, respectively. Next a is inserted into d, f, k, m, and the list reads a, d, f, k, m, b. Finally b is inserted into a, d, f, k, m, and the list reads a, b, d, f, k, m. At each insertion, the element to be inserted is compared with the elements already sorted, starting from the beginning, until its correct spot is found, and then the previously sorted elements beyond that spot are each moved one position toward the back of the list.

42. We let *minspot* be the place at which the minimum remaining element is found. After we find it on the i^{th} pass, we just have to interchange the elements in location *minspot* and location i.

```
procedure selection($a_1, a_2, \ldots, a_n$)
for $i := 1$ to $n - 1$
    $minspot := i$
    for $j := i + 1$ to $n$
        if $a_j < a_{minspot}$ then $minspot := j$
    interchange $a_{minspot}$ and $a_i$
{the list is now in order}
```

44. We carry out the binary search algorithm given as Algorithm 3 in this section, except that we replace the final check with **if** $x < a_i$ **then** *location* $:= i$ **else** *location* $:= i + 1$.

46. We are counting just the comparisons of the numbers in the list, not any comparisons needed for the bookkeeping in the **for** loop. The second element in the list must be compared only with the first (in other words, when $j = 2$ in Algorithm 5, i takes the values 1 before we drop out of the **while** loop). Similarly, the third element must be compared only with the first. We continue in this way, until finally the n^{th} element must be compared only with the first. So the total number of comparisons is $n - 1$. This is the best case for insertion sort in terms of the number of comparisons, but moving the elements to do the insertions requires much more effort.

48. For the insertion sort, one comparison is needed to find the correct location of the 4, one for the 3, four for the 8, one for the 1, four for the 5, and two for the 2. This is a total of 13 comparisons. For the binary insertion sort, one comparison is needed to find the correct location of the 4, two for the 3, two for the 8, three for the 1, three for the 5, and four for the 2. This is a total of 15 comparisons. If the list were long (and not almost in decreasing order to begin with), we would use many fewer comparisons using binary insertion sort. The reason that the answer came out "wrong" here is that the list is so short that the binary search was not efficient.

50. **a)** This is essentially the same as Algorithm 5, but working from the other end. However, we can do the moving while we do the searching for the correct insertion spot, so the pseudocode has only one section.

```
procedure backward insertion sort($a_1, a_2, \ldots, a_n$ : real numbers with $n \ge 2$)
for $j := 2$ to $n$
        $m := a_j$
        $i := j - 1$
        while ($m < a_i$ and $i > 0$)
                $a_{i+1} := a_i$
                $i := i - 1$
        $a_{i+1} := m$
{ $a_1, a_2, \ldots, a_n$ are sorted}
```

b) On the first pass the 2 is compared to the 3 and found to be less, so the 3 moves to the right. We have reached the beginning of the list, so the loop terminates ($i = 0$), and the 2 is inserted, yielding $2, 3, 4, 5, 1, 6$. On the second pass the 4 is compared to the 3, and since $4 > 3$, the **while** loop terminates and nothing changes. Similarly, no changes are made as the 5 is inserted. One the fourth pass, the 1 is compared all the way to the front of the list, with each element moving toward the back of the list as the comparisons go on, and finally the 1 is inserted in its correct position, yielding $1, 2, 3, 4, 5, 6$. The final pass produces no change.

c) Only one comparison is used during each pass, since the condition $m < a_i$ is immediately false. Therefore a total of $n - 1$ comparisons are used.

d) The j^{th} pass requires $j - 1$ comparisons of elements, so the total number of comparisons is $1 + 2 + \cdots + (n-1) = n(n-1)/2$.

52. In each case we use as many quarters as we can, then as many dimes to achieve the remaining amount, then as many nickels, then as many pennies.

a) The algorithm uses the maximum number of quarters, three, leaving 12 cents. It then uses the maximum number of dimes (one) and nickels (none), before using two pennies.

b) one quarter, leaving 24 cents, then two dimes, leaving 4 cents, then four pennies

c) three quarters, leaving 24 cents, then two dimes, leaving 4 cents, then four pennies

d) one quarter, leaving 8 cents, then one nickel and three pennies

54. **a)** The algorithm uses the maximum number of quarters, three, leaving 12 cents. It then uses the maximum number of dimes (one), and then two pennies. The greedy algorithm worked, since we got the same answer as in Exercise 52.

b) one quarter, leaving 24 cents, then two dimes, leaving 4 cents, then four pennies (the greedy algorithm worked, since we got the same answer as in Exercise 52)

c) three quarters, leaving 24 cents, then two dimes, leaving 4 cents, then four pennies (the greedy algorithm worked, since we got the same answer as in Exercise 52)

d) The greedy algorithm would have us use one quarter, leaving 8 cents, then eight pennies, a total of nine coins. However, we could have used three dimes and three pennies, a total of six coins. Thus the greedy algorithm is not correct for this set of coins.

56. One approach is to come up with an example in which using the 12-cent coin before using dimes or nickels would be inefficient. A dime and a nickel together are worth 15 cents, but the greedy algorithm would have us use four coins (a 12-cent coin and three pennies) rather than two. An alternative example would be 29 cents, in which case the greedy algorithm would use a quarter and four pennies, but we could have done better using two 12-cent coins and a nickel.

58. Here is one counterexample, using 11 talks. Suppose the start and end times are as follows: A 1–3, B 3–5, C 5–7, D 7–9, E 2–4, F 2–4, G 2–4, H 4–6, J 6–8, K 6–8, L 6–8. The optimal schedule is talks A, B, C, and D. However, the talk with the fewest overlaps with other talks is H, which overlaps only with B and C (all the other talks overlap with three or four other talks). However, once we have decided to include talk H, we can no longer schedule four talks, so this algorithm will not produce an optimum solution.

60. If all the men get their first choices, then the matching will be stable, because no man will be part of an unstable pair, preferring another woman to his assigned partner. Thus the pairing (m_1w_3, m_2w_1, m_3w_2) is stable. Similarly, if all the women get their first choices, then the matching will be stable, because no woman will be part of an unstable pair, preferring another man to her assigned partner. Thus the matching (m_1w_1, m_2w_2, m_3w_3) is stable. Two of the other four matchings pair m_1 with w_2, and this cannot be stable, because m_1 prefers w_1 to w_2, his assigned partner, and w_1 prefers m_1 to her assigned partner, whoever it is, because m_1 is her favorite. In a similar way, the matching (m_1w_3, m_2w_2, m_3w_1) is unstable because of the unhappy unmatched pair m_3w_3 (each preferring the other to his or her assigned partner). Finally, the matching (m_1w_1, m_2w_3, m_3w_2) is stable, because each couple has a reason not to break up: w_1 got her favorite and so is content, m_3 got his favorite and so is content, and w_3 only prefers m_3 to her assigned partner but he doesn't prefer her to his assigned partner.

62. The algorithm given in the solution to Exercise 61 will terminate if at some point at the conclusion of the **while** loop, no man is rejected. If this happens, then that must mean that each man has one and only one proposal pending with some woman, because he proposed to only one in that round, and since he was not rejected, his proposal is the only one pending with that woman. It follows that at that point there are s pending proposals, one from each man, so each woman will be matched with a unique man. Finally, we argue that there are at most s^2 iterations of the **while** loop, so the algorithm must terminate. Indeed, if at the conclusion of the **while** loop rejected men remain, then some man must have been rejected, because no man is marked as rejected at the conclusion of the proposal phase (first **for** loop inside the **while** loop). If a man is rejected, then his rejection list grows. Thus each pass through the **while** loop, at least one more of the s^2 possible rejections will have been recorded, unless the loop is about to terminate. (Actually there will be fewer than s^2 iterations, because no man is rejected by the woman with whom he is eventually matched.) There is one more subtlety we need to address. Is it possible that at the end of some round, some man has been rejected by *every* woman and therefore the algorithm cannot continue? We claim not. If at the end of some round some man has been rejected by every woman, then every woman has one pending proposal at the completion of that round (from someone she likes better—otherwise she never would have rejected that poor man), and of course these proposals are all from different men because a man proposes only once in each round. That means s men have pending proposals, so in fact our poor universally-rejected man does not exist.

64. Suppose we had a program S that could tell whether a program with its given input ever prints the digit 1. Here is an algorithm for solving the halting problem: Given a program P and its input I, construct a program P', which is just like P but never prints anything (even if P did print something) except that if and when it is about to halt, it prints a 1 and halts. Then P halts on an input if and only if P' ever prints a 1 on that same input. Feed P' and I to S, and that will tell us whether or not P halts on input I. Since we know that the halting problem is in fact not solvable, we have a contradiction. Therefore no such program S exists.

66. The decision problem has no input. The answer is either always yes or always no, depending on whether or not the specific program with its specific input halts or not. In the former case, the decision procedure is "say yes," and in the latter case it is "say no."

SECTION 3.2 The Growth of Functions

2. Note that the choices of C and k witnesses are not unique.

a) Yes, since $17x + 11 \leq 17x + x = 18x \leq 18x^2$ for all $x > 11$. The witnesses are $C = 18$ and $k = 11$.

b) Yes, since $x^2 + 1000 \leq x^2 + x^2 = 2x^2$ for all $x > \sqrt{1000}$. The witnesses are $C = 2$ and $k = \sqrt{1000}$.

c) Yes, since $x \log x \leq x \cdot x = x^2$ for all x in the domain of the function. (The fact that $\log x < x$ for all x follows from the fact that $x < 2^x$ for all x, which can be seen by looking at the graphs of these two functions.) The witnesses are $C = 1$ and $k = 0$.

d) No. If there were a constant C such that $x^4/2 \leq Cx^2$ for sufficiently large x, then we would have $C \geq x^2/2$. This is clearly impossible for a constant to satisfy.

e) No. If 2^x were $O(x^2)$, then the fraction $2^x/x^2$ would have to be bounded above by some constant C. It can be shown that in fact $2^x > x^3$ for all $x \geq 10$ (using mathematical induction—see Section 5.1—or calculus), so $2^x/x^2 \geq x^3/x^2 = x$ for large x, which is certainly not less than or equal to C.

f) Yes, since $\lfloor x \rfloor \lceil x \rceil \leq x(x+1) \leq x \cdot 2x = 2x^2$ for all $x > 1$. The witnesses are $C = 2$ and $k = 1$.

4. If $x > 5$, then $2^x + 17 \leq 2^x + 2^x = 2 \cdot 2^x \leq 2 \cdot 3^x$. This shows that $2^x + 17$ is $O(3^x)$ (the witnesses are $C = 2$ and $k = 5$).

6. We can use the following inequalities, valid for all $x > 1$ (note that making the denominator of a fraction smaller makes the fraction larger).

$$\frac{x^3 + 2x}{2x + 1} \leq \frac{x^3 + 2x^3}{2x} = \frac{3}{2}x^2$$

This proves the desired statement, with witnesses $k = 1$ and $C = 3/2$.

8. **a)** Since $x^3 \log x$ is not $O(x^3)$ (because the $\log x$ factor grows without bound as x increases), $n = 3$ is too small. On the other hand, certainly $\log x$ grows more slowly than x, so $2x^2 + x^3 \log x \leq 2x^4 + x^4 = 3x^4$. Therefore $n = 4$ is the answer, with $C = 3$ and $k = 0$.

b) The $(\log x)^4$ is insignificant compared to the x^5 term, so the answer is $n = 5$. Formally we can take $C = 4$ and $k = 1$ as witnesses.

c) For large x, this fraction is fairly close to 1. (This can be seen by dividing numerator and denominator by x^4.) Therefore we can take $n = 0$; in other words, this function is $O(x^0) = O(1)$. Note that $n = -1$ will not do, since a number close to 1 is not less than a constant times n^{-1} for large n. Formally we can write $f(x) \leq 3x^4/x^4 = 3$ for all $x > 1$, so witnesses are $C = 3$ and $k = 1$.

d) This is similar to the previous part, but this time $n = -1$ will do, since for large x, $f(x) \approx 1/x$. Formally we can write $f(x) \leq 6x^3/x^3 = 6$ for all $x > 1$, so witnesses are $C = 6$ and $k = 1$.

10. Since $x^3 \leq x^4$ for all $x > 1$, we know that x^3 is $O(x^4)$ (witnesses $C = 1$ and $k = 1$). On the other hand, if $x^4 \leq Cx^3$, then (dividing by x^3) $x \leq C$. Since this latter condition cannot hold for all large x, no matter what the value of the constant C, we conclude that x^4 is not $O(x^3)$.

12. We showed that $x \log x$ is $O(x^2)$ in Exercise 2c. To show that x^2 is not $O(x \log x)$ it is enough to show that $x^2/(x \log x)$ is unbounded. This is the same as showing that $x/\log x$ is unbounded. First let us note that $\log x < \sqrt{x}$ for all $x > 16$. This can be seen by looking at the graphs of these functions, or by calculus. Therefore the fraction $x/\log x$ is greater than $x/\sqrt{x} = \sqrt{x}$ for all $x > 16$, and this clearly is not bounded.

14. **a)** No, by an argument similar to Exercise 10.
b) Yes, since $x^3 \le x^3$ for all x (witnesses $C = 1,\ k = 0$).
c) Yes, since $x^3 \le x^2 + x^3$ for all x (witnesses $C = 1,\ k = 0$).
d) Yes, since $x^3 \le x^2 + x^4$ for all x (witnesses $C = 1,\ k = 0$).
e) Yes, since $x^3 \le 2^x \le 3^x$ for all $x > 10$ (see Exercise 2e). Thus we have witnesses $C = 1$ and $k = 10$.
f) Yes, since $x^3 \le 2 \cdot (x^3/2)$ for all x (witnesses $C = 2,\ k = 0$).

16. The given information says that $|f(x)| \le C|x|$ for all $x > k$, where C and k are particular constants. Let k' be the larger of k and 1. Then since $|x| \le |x^2|$ for all $x > 1$, we have $|f(x)| \le C|x^2|$ for all $x > k'$, as desired.

18. $1^k + 2^k + \cdots + n^k \le n^k + n^k + \cdots + n^k = n \cdot n^k = n^{k+1}$

20. They both are. For the first we have $\log(n+1) < \log(2n) = \log n + \log 2 < 2\log n$ for $n > 2$. For the second one we have $\log(n^2+1) < \log(2n^2) = 2\log n + \log 2 < 3\log n$ for $n > 2$.

22. The ordering is straightforward when we remember that exponential functions grow faster than polynomial functions, that factorial functions grow faster still, and that logarithmic functions grow very slowly. The order is $(\log n)^3$, $\sqrt{n}\log n$, $n^{99} + n^{98}$, n^{100}, 1.5^n, 10^n, $(n!)^2$.

24. The first algorithm uses fewer operations because $n^2 2^n$ is $O(n!)$ but $n!$ is not $O(n^2 2^n)$. In fact, the second function overtakes the first function for good at $n = 8$, when $8^2 \cdot 2^8 = 16{,}384$ and $8! = 40{,}320$.

26. The approach in these problems is to pick out the most rapidly growing term in each sum and discard the rest (including the multiplicative constants).
a) This is $O(n^3 \cdot \log n + \log n \cdot n^3)$, which is the same as $O(n^3 \cdot \log n)$.
b) Since 2^n dominates n^2, and 3^n dominates n^3, this is $O(2^n \cdot 3^n) = O(6^n)$.
c) The dominant terms in the two factors are n^n and $n!$, respectively. Therefore this is $O(n^n n!)$.

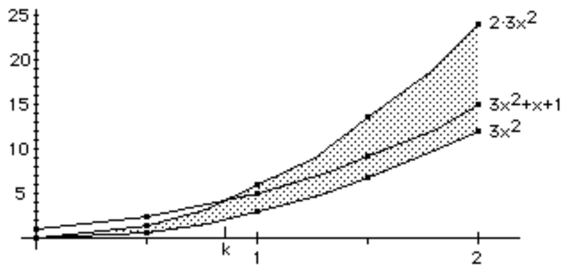
28. We can use the following rule of thumb to determine what simple big-Theta function to use: throw away all the lower order terms (those that don't grow as fast as other terms) and all constant coefficients.
a) This function is $\Theta(1)$, so it is not $\Theta(x)$, since 1 (or 10) grows more slowly than x. To be precise, x is not $O(10)$. For the same reason, this function is not $\Omega(x)$.
b) This function is $\Theta(x)$; we can ignore the " $+\ 7$" since it is a lower order term, and we can ignore the coefficient. Of course, since $f(x)$ is $\Theta(x)$, it is also $\Omega(x)$.
c) This function grows faster than x. Therefore $f(x)$ is not $\Theta(x)$ but it is $\Omega(x)$.
d) This function grows more slowly than x. Therefore $f(x)$ is not $\Theta(x)$ or $\Omega(x)$.
e) This function has values that are, for all practical purposes, equal to x (certainly $\lfloor x \rfloor$ is always between $x/2$ and x, for $x > 2$), so it is $\Theta(x)$ and therefore also $\Omega(x)$.
f) As in part **(e)** this function has values that are, for all practical purposes, equal to $x/2$, so it is $\Theta(x)$ and therefore also $\Omega(x)$.

30. **a)** This follows from the fact that for all $x > 7$, $x \le 3x + 7 \le 4x$.
b) For large x, clearly $x^2 \le 2x^2 + x - 7$. On the other hand, for $x \ge 1$ we have $2x^2 + x - 7 \le 3x^2$.
c) For $x > 2$ we certainly have $\lfloor x + \frac{1}{2} \rfloor \le 2x$ and also $x \le 2\lfloor x + \frac{1}{2} \rfloor$.
d) For $x > 2$, $\log(x^2 + 1) \le \log(2x^2) = 1 + 2\log x \le 3\log x$ (recall that $\log$ means $\log_2$). On the other hand, since $x < x^2 + 1$ for all positive x, we have $\log x \le \log(x^2 + 1)$.
e) This follows from the fact that $\log_{10} x = C(\log_2 x)$, where $C = 1/\log_2 10$.

32. We just need to look at the definitions. To say that $f(x)$ is $O(g(x))$ means that there are constants C and k such that $|f(x)| \leq C|g(x)|$ for all $x > k$. Note that without loss of generality we may take C and k to be positive. To say that $g(x)$ is $\Omega(f(x))$ is to say that there are positive constants C' and k' such that $|g(x)| \geq C'|f(x)|$ for all $x > k$. These are saying exactly the same thing if we set $C' = 1/C$ and $k' = k$.

34. **a)** By Exercise 31 we have to show that $3x^2 + x + 1$ is $O(3x^2)$ and that $3x^2$ is $O(3x^2 + x + 1)$. The latter is trivial, since $3x^2 \leq 3x^2+x+1$ for $x > 0$. The former is almost as trivial, since $3x^2+x+1 \leq 3x^2+3x^2 = 2\cdot 3x^2$ for all $x > 1$. What we have shown is that $1\cdot 3x^2 \leq 3x^2+x+1 \leq 2\cdot 3x^2$ for all $x > 1$; in other words, $C_1 = 1$ and $C_2 = 2$ in Exercise 33.

b) The following picture shows that graph of $3x^2 + x + 1$ falls in the shaded region between the graph of $3x^2$ and the graph of $2 \cdot 3x^2$ for all $x > 1$.



36. Looking at the definition, we see that to say that $f(x)$ is $\Omega(1)$ means that $|f(x)| \geq C$ when $x > k$, for some positive constants k and C. In other words, $f(x)$ keeps at least a certain distance away from 0 for large enough x. For example, $1/x$ is not $\Omega(1)$, since it gets arbitrary close to 0; but $(x-2)(x-10)$ is $\Omega(1)$, since $f(x) \geq 9$ for $x > 11$.

38. The n^{th} odd positive integer is $2n-1$. Thus each of the first n odd positive integers is at most $2n$. Therefore their product is at most $(2n)^n$, so one answer is $O\big((2n)^n\big)$. Of course other answers are possible as well.

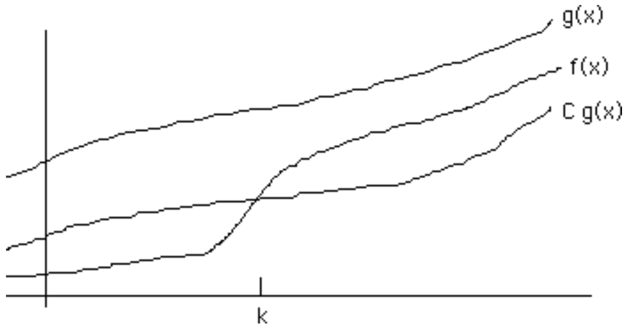
40. This follows from the fact that $\log_b x$ and $\log_a x$ are the same except for a multiplicative constant, namely $d = \log_b a$. Thus if $f(x) \leq C \log_b x$, then $f(x) \leq Cd \log_a x$.

42. This does not follow. Let $f(x) = 2x$ and $g(x) = x$. Then $f(x)$ is $O(g(x))$. Now $2^{f(x)} = 2^{2x} = 4^x$, and $2^{g(x)} = 2^x$, and 4^x is not $O(2^x)$. Indeed, $4^x/2^x = 2^x$, so the ratio grows without bound as x grows—it is not bounded by a constant.

44. The definition of "$f(x)$ is $\Theta(g(x))$" is that $f(x)$ is both $O(g(x))$ and $\Omega(g(x))$. That means that there are positive constants C_1, k_1, C_2, and k_2 such that $|f(x)| \leq C_2|g(x)|$ for all $x > k_2$ and $|f(x)| \geq C_1|g(x)|$ for all $x > k_1$. Similarly, we have that there are positive constants C_1', k_1', C_2', and k_2' such that $|g(x)| \leq C_2'|h(x)|$ for all $x > k_2'$ and $|g(x)| \geq C_1'|h(x)|$ for all $x > k_1'$. We can combine these inequalities to obtain $|f(x)| \leq C_2C_2'|h(x)|$ for all $x > \max(k_2, k_2')$ and $|f(x)| \geq C_1C_1'|h(x)|$ for all $x > \max(k_1, k_1')$. This means that $f(x)$ is $\Theta(h(x))$.

46. The definitions tell us that there are positive constants C_1, k_1, C_2, and k_2 such that $|f_1(x)| \leq C_2|g_1(x)|$ for all $x > k_2$ and $|f_1(x)| \geq C_1|g_1(x)|$ for all $x > k_1$, and that there are positive constants C_1', k_1', C_2', and k_2' such that $|f_2(x)| \leq C_2'|g_2(x)|$ for all $x > k_2'$ and $|f_2(x)| \geq C_1'|g_2(x)|$ for all $x > k_1'$. We can multiply these inequalities to obtain $|f_1(x)f_2(x)| \leq C_2C_2'|g_1(x)g_2(x)|$ for all $x > \max(k_2, k_2')$ and $|f_1(x)f_2(x)| \geq C_1C_1'|g_1(x)g_2(x)|$ for all $x > \max(k_1, k_1')$. This means that $f_1(x)f_2(x)$ is $\Theta(g_1(x)g_2(x))$.

48. Typically C will be less than 1. From some point onward to the right ($x > k$), the graph of $f(x)$ must be above the graph of $g(x)$ after the latter has been scaled down by the factor C. Note that $f(x)$ does not have to be larger than $g(x)$ itself.



50. We need to show inequalities both ways. First, we show that $|f(x)| \le Cx^n$ for all $x \ge 1$, as follows, noting that $x^i \le x^n$ for such values of x whenever $i < n$. We have the following inequalities, where M is the largest of the absolute values of the coefficients and C is $M(n+1)$:

$$\begin{aligned}|f(x)| &= |a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0|\\ &\le |a_n|x^n + |a_{n-1}|x^{n-1} + \cdots + |a_1|x + |a_0|\\ &\le |a_n|x^n + |a_{n-1}|x^n + \cdots + |a_1|x^n + |a_0|x^n\\ &\le Mx^n + Mx^n + \cdots + Mx^n + Mx^n = Cx^n\end{aligned}$$

For the other direction, which is a little messier, let k be chosen larger than 1 and larger than $2nm/|a_n|$, where m is the largest of the absolute values of the a_i's for $i < n$. Then each a_{n-i}/x^i will be smaller than $|a_n|/2n$ in absolute value for all $x > k$. Now we have for all $x > k$,

$$\begin{aligned}|f(x)| &= |a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0|\\ &= x^n \left|a_n + \frac{a_{n-1}}{x} + \cdots + \frac{a_1}{x^{n-1}} + \frac{a_0}{x^n}\right|\\ &\ge x^n\,|a_n/2|\,,\end{aligned}$$

as desired.

52. We just make the analogous change in the definition of big-Omega that was made in the definition of big-O: there exist positive constants C, k_1, and k_2 such that $|f(x,y)| \ge C|g(x,y)|$ for all $x > k_1$ and $y > k_2$.

54. For all values of x and y greater than 1, each term of the given expression is greater than x^3y^3, so the entire expression is greater than x^3y^3. In other words, we take $C = k_1 = k_2 = 1$ in the definition given in Exercise 52.

56. For all positive values of x and y, we know that $\lceil xy \rceil \ge xy$ by definition (since the ceiling function value cannot be less than the argument). Thus $\lceil xy \rceil$ is $\Omega(xy)$ from the definition, taking $C = 1$ and $k_1 = k_2 = 0$. In fact, $\lceil xy \rceil$ is also $O(xy)$ (and therefore $\Theta(xy)$); this is easy to see since $\lceil xy \rceil \le (x+1)(y+1) \le (2x)(2y) = 4xy$ for all x and y greater than 1.

58. It suffices to show that

$$\lim_{n\to\infty} \frac{(\log_b n)^c}{n^d} = 0\,,$$

where we think of n as a continuous variable. Because both numerator and denominator approach ∞, we apply L'Hôpital's rule and evaluate

$$\lim_{n\to\infty} \frac{c(\log_b n)^{c-1}}{d \cdot n^d \cdot \ln b}\,.$$

At this point, if $c \leq 1$, then the limit is 0. Otherwise we again have an expression of type ∞/∞, so we apply L'Hôpital's rule once more, obtaining

$$\lim_{n\to\infty} \frac{c(c-1)(\log_b n)^{c-2}}{d^2 \cdot n^d \cdot (\ln b)^2}\,.$$

If $c \leq 2$, then the limit is 0; if not, we repeat. Eventually the exponent on $\log_b n$ becomes nonpositive and we conclude that the limit is 0, as desired.

60. If suffices to look at $\lim_{n\to\infty} b^n/c^n = (b/c)^n$ and $\lim_{n\to\infty} c^n/b^n = (c/b)^n$. Because $c > b > 1$, we have $0 < b/c < 1$ and $c/b > 1$, so the former limit is clearly 0 and the latter limit is clearly ∞.

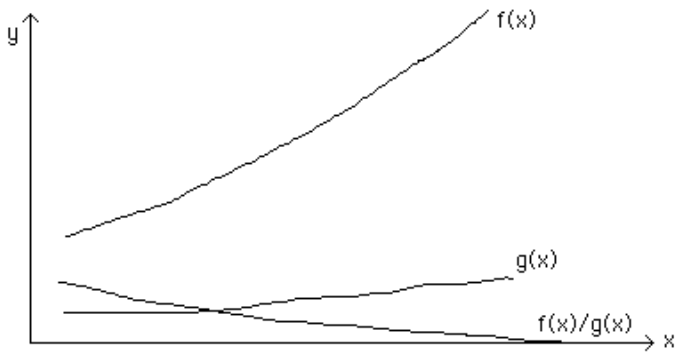
62. a) Under the hypotheses,

$$\lim_{x\to\infty} \frac{cf(x)}{g(x)} = c \lim_{x\to\infty} \frac{f(x)}{g(x)} = c \cdot 0 = 0\,.$$

b) Under the hypotheses,

$$\lim_{x\to\infty} \frac{f_1(x)+f_2(x)}{g(x)} = \lim_{x\to\infty} \frac{f_1(x)}{g(x)} + \lim_{x\to\infty} \frac{f_2(x)}{g(x)} = 0 + 0 = 0\,.$$

64. The behaviors of f and g alone are not really at issue; what is important is whether $f(x)/g(x)$ approaches 0 as $x \to \infty$. Thus, as shown in the picture, it might happen that the graphs of f and g rise, but f increases enough more rapidly than g so that the ratio gets small. In the picture, we see that $f(x)/g(x)$ is asymptotic to the x-axis.



66. No. Let $f(x) = x$ and $g(x) = x^2$. Then clearly $f(x)$ is $o(g(x))$, but the ratio of the logs of the absolute values is the constant 2, and 2 does not approach 0. Therefore it is not the case in this example that $\log|f(x)|$ is $o(\log|g(x)|)$.

68. This follows from the fact that the limit of $f(x)/g(x)$ is 0 in this case, as can be most easily seen by dividing numerator and denominator by x^n (the numerator then is bounded and the absolute value of the denominator grows without bound as $x \to \infty$).

70. Since $f(x) = 1/x$ is a decreasing function which has the value $1/x$ at $x = j$, it is clear that $1/j < 1/x$ throughout the interval from $j-1$ to j. Summing over all the intervals for $j = 2, 3, \ldots, n$, and noting that the definite integral is the area under the curve, we obtain the inequality in the hint. Therefore

$$H_n = 1 + \sum_{j=2}^{n} \frac{1}{j} < 1 + \int_1^n \frac{1}{x}\,dx = 1 + \ln n = 1 + C\log n \leq 2C\log n$$

for $n > 2$, where $C = \log e$.

72. By Example 6, $\log n!$ is $O(n\log n)$. By Exercise 71, $n\log n$ is $O(\log n!)$. Thus by Exercise 31, $\log n!$ is $\Theta(n\log n)$.

74. In each case we need to evaluate the limit of $f(x)/g(x)$ as $x\to\infty$. If it equals 1, then f and g are asymptotic; otherwise (including the case in which the limit does not exist) they are not. Most of these are straightforward applications of algebra, elementary notions about limits, or L'Hôpital's rule.

a) $\lim\limits_{x\to\infty}\dfrac{x^2+3x+7}{x^2+10}=\lim\limits_{x\to\infty}\dfrac{1+3/x+7/x^2}{1+10/x^2}=1$, so f and g are asymptotic.

b) $\lim\limits_{x\to\infty}\dfrac{x^2\log x}{x^3}=\lim\limits_{x\to\infty}\dfrac{\log x}{x}=\lim\limits_{x\to\infty}\dfrac{1}{x\cdot\ln 2}=0$ (we used L'Hôpital's rule for the last equivalence), so f and g are not asymptotic.

c) Here $f(x)$ is dominated by its leading term, x^4, and $g(x)$ is a polynomial of degree 4, so the ratio approaches 1, the ratio of the leading coefficients, as in part **(a)**. Therefore f and g are asymptotic.

d) Here f and g are polynomials of degree 12, so the ratio approaches 1, the ratio of the leading coefficients, as in part **(a)**. Therefore f and g are asymptotic.

SECTION 3.3 Complexity of Algorithms

2. The statement $t := t+i+j$ is executed n^2 times, so the number of operations is $O(n^2)$. (Specifically, $2n^2$ additions are used, not counting any arithmetic needed for bookkeeping in the loops.)

4. The value of i keeps doubling, so the loop terminates after k iterations as soon as $2^k > n$. The value of k that makes this happen is $O(\log n)$, because $2^{\log n}=n$. Within the loop there are two additions or multiplications, so the answer to the question is $O(\log n)$.

6. **a)** We can sort the first four elements by copying the steps in Algorithm 5 but only up to $j=4$.

```
procedure sort four($a_1, a_2, \ldots, a_n$ : real numbers)
for $j := 2$ to 4
        $i := 1$
        while $a_j > a_i$
                $i := i + 1$
        $m := a_j$
        for $k := 0$ to $j - i - 1$
                $a_{j-k} := a_{j-k-1}$
        $a_i := m$
```

b) Only a (small) finite number of steps are performed here, regardless of the length of the list, so this algorithm has complexity $O(1)$.

8. If we successively square k times, then we have computed x^{2^k}. Thus we can compute x^{2^k} with only k multiplications, rather than the 2^k-1 multiplications that the naive algorithm would require, so this method is much more efficient.

10. **a)** By the way that $S-1$ is defined, it is clear that $S\wedge(S-1)$ is the same as S except that the rightmost 1 bit has been changed to a 0. Thus we add 1 to *count* for every one bit (since we stop as soon as $S=0$, i.e., as soon as S consists of just 0 bits).

b) Obviously the number of bitwise *AND* operations is equal to the final value of *count*, i.e., the number of one bits in S.

12. **a)** There are three loops, each nested inside the next. The outer loop is executed n times, the middle loop is executed at most n times, and the inner loop is executed at most n times. Therefore the number of times the one statement inside the inner loop is executed is at most n^3. This statement requires one comparison, so the total number of comparisons is $O(n^3)$.

b) We follow the hint, not worrying about the fractions that might result from roundoff when dividing by 2 or 4 (these don't affect the final answer in big-Omega terms). The outer loop is executed at least $n/4$ times, once for each value of i from 1 to $n/4$ (we ignore the rest of the values of i). The middle loop is executed at least $n/4$ times, once for each value of j from $3n/4$ to n. The inner loop for these values of i and j is executed at least $(3n/4) - (n/4) = n/2$ times. Therefore the statement within the inner loop, which requires one comparison, is executed at least $(n/4)(n/4)(n/2) = n/32$ times, which is $\Omega(n^3)$. The second statement follows by definition.

14. **a)** Initially $y := 3$. For $i = 1$ we set y to $3 \cdot 2 + 1 = 7$. For $i = 2$ we set y to $7 \cdot 2 + 1 = 15$, and we are done.

b) There is one multiplication and one addition for each of the n passes through the loop, so there are n multiplications and n additions in all.

16. If each bit operation takes 10^{-11} second, then we can carry out 10^{11} bit operations per second, and therefore $60 \cdot 60 \cdot 24 \cdot 10^{11} = 864 \cdot 10^{13}$ bit operations per day. Therefore in each case we want to solve the inequality $f(n) = 864 \cdot 10^{13}$ for n and round down to an integer. Obviously a calculator or computer software will come in handy here.

a) If $\log n = 864 \cdot 10^{13}$, then $n = 2^{864 \cdot 10^{13}}$, which is an unfathomably huge number.

b) If $1000n = 864 \cdot 10^{13}$, then $n = 864 \cdot 10^{10}$, which is still a very large number.

c) If $n^2 = 864 \cdot 10^{13}$, then $n = \sqrt{864 \cdot 10^{13}}$, which works out to about $9.3 \cdot 10^7$.

d) If $1000n^2 = 864 \cdot 10^{13}$, then $n = \sqrt{864 \cdot 10^{10}}$, which works out to about $2.9 \cdot 10^6$.

e) If $n^3 = 864 \cdot 10^{13}$, then $n = (864 \cdot 10^{13})^{1/3}$, which works out to about $2.1 \cdot 10^5$.

f) If $2^n = 864 \cdot 10^{13}$, then $n = \lfloor \log(864 \cdot 10^{13}) \rfloor = 52$. (Remember, we are taking log to the base 2.)

g) If $2^{2n} = 864 \cdot 10^{13}$, then $n = \lfloor \log(864 \cdot 10^{13})/2 \rfloor = 26$.

h) If $2^{2^n} = 864 \cdot 10^{13}$, then $n = \lfloor \log(\log(864 \cdot 10^{13})) \rfloor = 5$.

18. We are asked to compute $(2n^2 + 2^n) \cdot 10^{-9}$ for each of these values of n. When appropriate, we change the units from seconds to some larger unit of time.

a) 1.224×10^{-6} seconds **b)** approximately 1.05×10^{-3} seconds

c) approximately 1.13×10^6 seconds, which is about 13 days (nonstop)

d) approximately 1.27×10^{21} seconds, which is about 4×10^{13} years (nonstop)

20. In each case we want to compare the function evaluated at $2n$ to the function evaluated at n. The most desirable form of the comparison (subtraction or division) will vary.

a) Notice that

$$\log\log 2n - \log\log n = \log \frac{\log 2 + \log n}{\log n} = \log \frac{1 + \log n}{\log n}\,.$$

If n is large, the fraction in this expression is approximately equal to 1, and therefore the expression is approximately equal to 0. In other words, hardly any extra time is required. For example, in going from $n = 1024$ to $n = 2048$, the number of extra milliseconds is $\log 11/10 \approx 0.14$.

b) Here we have $\log 2n - \log n = \log \frac{2n}{n} = \log 2 = 1$. One extra millisecond is required, independent of n.

c) This time it makes more sense to use a ratio comparison, rather than a difference comparison. Because $100(2n)/(100n) = 2$, we conclude that twice as much time is needed for the larger problem.

d) The controlling factor here is n, rather than $\log n$, so again we look at the ratio:

$$\frac{2n\log(2n)}{n\log n} = 2\cdot\frac{1+\log n}{\log n}$$

For large n, the final fraction is approximately 1, so we can say that the time required for $2n$ is a bit more than twice what it is for n.

e) Because $(2n)^2/n^2 = 4$, we see that four times as much time is required for the larger problem.

f) Because $(3n)^2/n^2 = 9$, we see that nine times as much time is required for the larger problem.

g) The relevant ratio is $2^{2n}/2^n$, which equals 2^n. If n is large, then this is a huge number. For example, in going from $n = 10$ to $n = 20$, the number of milliseconds increases over 1000-fold.

22. **a)** The number of comparisons does not depend on the values of a_1 through a_n. Exactly $2n-1$ comparisons are used, as was determined in Example 1. In other words, the best case performance is $O(n)$.

b) In the best case $x = a_1$. We saw in Example 4 that three comparisons are used in that case. The best case performance, then, is $O(1)$.

c) It is hard to give an exact answer, since it depends on the binary representation of the number n, among other things. In any case, the best case performance is really not much different from the worst case performance, namely $O(\log n)$, since the list is essentially cut in half at each iteration, and the algorithm does not stop until the list has only one element left in it.

24. **a)** In order to find the maximum element of a list of n elements, we need to make at least $n-1$ comparisons, one to rule out each of the other elements. Since Algorithm 1 in Section 3.1 used just this number (not counting bookkeeping), it is optimal.

b) Linear search is not optimal, since we found that binary search was more efficient. This assumes that we can be given the list already sorted into increasing order.

26. We will count comparisons of elements in the list to x. (This ignores comparisons of subscripts, but since we are only interested in a big-O analysis, no harm is done.) Furthermore, we will assume that the number of elements in the list is a power of 4, say $n = 4^k$. Just as in the case of binary search, we need to determine the maximum number of times the **while** loop is iterated. Each pass through the loop cuts the number of elements still being considered (those whose subscripts are from i to j) by a factor of 4. Therefore after k iterations, the active portion of the list will have length 1; that is, we will have $i = j$. The loop terminates at this point. Now each iteration of the loop requires two comparisons in the worst case (one with a_m and one with either a_l or a_u). Three more comparisons are needed at the end. Therefore the number of comparisons is $2k+3$, which is $O(k)$. But $k = \log_4 n$, which is $O(\log n)$ since logarithms to different bases differ only by multiplicative constants, so the time complexity of this algorithm (in all cases, not just the worst case) is $O(\log n)$.

28. The algorithm we gave for finding all the modes essentially just goes through the list once, doing a little bookkeeping at each step. In particular, between any two successive executions of the statement $i := i+1$ there are at most about eight operations (such as comparing *count* with *modecount*, or reinitializing *value*). Therefore at most about $8n$ steps are done in all, so the time complexity in all cases is $O(n)$.

30. The algorithm we gave is clearly of linear time complexity, i.e., $O(n)$, since we were able to keep updating the sum of previous terms, rather than recomputing it each time. This applies in all cases, not just the worst case.

32. The algorithm read through the list once and did a bounded amount of work on each term. Looked at another way, only a bounded amount of work was done between increments of j in the algorithm given in the solution. Thus the complexity is $O(n)$.

34. It takes $n-1$ comparisons to find the least element in the list, then $n-2$ comparisons to find the least element among the remaining elements, and so on. Thus the total number of comparisons is $(n-1)+(n-2)+\cdots+2+1 = n(n-1)/2$, which is $O(n^2)$.

36. Each iteration (determining whether we can use a coin of a given denomination) takes a bounded amount of time, and there are at most n iterations, since each iteration decreases the number of cents remaining. Therefore there are $O(n)$ comparisons.

38. First we sort the talks by earliest end time; this takes $O(n\log n)$ time if there are n talks. We initialize a variable *opentime* to be 0; it will be updated whenever we schedule another talk to be the time at which that talk ends. Next we go through the list of talks in order, and for each talk we see whether its start time does not precede *opentime* (we already know that its ending time exceeds *opentime*). If so, then we schedule that talk and update *opentime* to be its ending time. This all takes $O(1)$ time per talk, so the entire process after the initial sort has time complexity $O(n)$. Combining this with the initial sort, we get an overall time complexity of $O(n\log n)$.

40. **a)** The bubble sort algorithm uses about $n^2/2$ comparisons for a list of length n, and $(2n)^2/2 = 2n^2$ comparisons for a list of length $2n$. Therefore the number of comparisons goes up by a factor of 4.

b) The analysis is the same as for bubble sort.

c) The analysis is the same as for bubble sort.

d) The binary insertion sort algorithm uses about $Cn\log n$ comparisons for a list of length n, where C is a constant. Therefore it uses about $C\cdot 2n\log 2n = C\cdot 2n\log 2 + C\cdot 2n\log n = C\cdot 2n + C\cdot 2n\log n$ comparisons for a list of length $2n$. Therefore the number of comparisons increases by about a factor of 2 (for large n, the first term is small compared to the second and can be ignored).

42. In an $n\times n$ upper-triangular matrix, all entries a_{ij} are zero unless $i\le j$. Therefore we can store such matrices in about half the space that would be required to store an ordinary $n\times n$ matrix. In implementing something like Algorithm 1, then, we need only do the computations for those values of the indices that can produce nonzero entries. The following algorithm does this. We follow the usual notation: $\mathbf{A}=[a_{ij}]$ and $\mathbf{B}=[b_{ij}]$.

```
procedure triangular matrix multiplication($\mathbf{A}, \mathbf{B}$ : upper-triangular matrices)
for $i := 1$ to $n$
    for $j := i$ to $n$ {since we want $j \ge i$}
        $c_{ij} := 0$
        for $k := i$ to $j$ {the only relevant part}
            $c_{ij} := c_{ij} + a_{ik}b_{kj}$
{the upper-triangular matrix $\mathbf{C} = [c_{ij}]$ is the product of $\mathbf{A}$ and $\mathbf{B}$}
```

44. We have two choices: $(\mathbf{AB})\mathbf{C}$ or $\mathbf{A}(\mathbf{BC})$. For the first choice, it takes $3\cdot 9\cdot 4 = 144$ multiplications to form the 3×4 matrix $\mathbf{AB}$, and then $3\cdot 4\cdot 2 = 24$ multiplications to get the final answer, for a total of 168 multiplications. For the second choice, it takes $9\cdot 4\cdot 2 = 72$ multiplications to form the 9×2 matrix $\mathbf{BC}$, and then $3\cdot 9\cdot 2 = 54$ multiplications to get the final answer, for a total of 126 multiplications. The second method uses fewer multiplications and so is the better choice.

46. **a)** Let us call the text $s_1s_2\ldots s_n$ and call the target $t_1t_2\ldots t_m$. We want to find the first occurrence of $t_1t_2\ldots t_m$ in $s_1s_2\ldots s_n$, which means we want to find the smallest $k\ge 0$ such that $t_1t_2\ldots t_m = s_{k+1}s_{k+2}\ldots s_{k+m}$. The brute force algorithm will try $k=0,1,\ldots,n-m$ and for each such k check whether $t_j = s_{k+j}$ for $j=1,2,\ldots,m$. If these equalities all hold, the value $k+1$ will be returned (that's where the target starts); otherwise 0 will be returned (as a code for "not there").

b) The implementation is straightforward:

```
procedure findit($s_1s_2\dots s_n, t_1t_2\dots t_m$ : strings)
found := false
$k := 0$
while $k \le m - n$ and not found
    found := true
    for $j := i$ to $m$
        if $t_j \ne s_{k+j}$ then found := false
    if found then return $k + 1$ {location of start of target $t_1t_2\dots t_m$ in text $s_1s_2\dots s_n$ }
return 0 {target $t_1t_2\dots t_m$ does not appear in text $s_1s_2\dots s_n$ }
```

c) Because of the nested loops, the worst-case time complexity will be $O(mn)$.

SUPPLEMENTARY EXERCISES FOR CHAPTER 3

2. **a)** We need to keep track of the first and second largest elements as we go along, updating as we look at the elements in the list.

```
procedure toptwo($a_1, a_2, \dots, a_n$ : integers)
largest := $a_1$
second := $-\infty$
for $i := 2$ to $n$
    if $a_i >$ second then second := $a_i$
    if $a_i >$ largest then
        second := largest
        largest := $a_i$
{ largest and second are the required values}
```

b) The loop is executed $n - 1$ times, and there are 2 comparisons per iteration. Therefore (ignoring bookkeeping) there are $2n - 2$ comparisons.

4. **a)** Since the list is in order, all the occurrences appear consecutively. Thus the output of our algorithm will be a pair of numbers, *first* and *last*, which give the first location and the last location of occurrences of x, respectively. All the numbers between *first* and *last* are also locations of appearances of x. If there are no appearances of x, we set *first* equal to 0 to indicate this fact.

```
procedure all($x, a_1, a_2, \dots, a_n$ : integers, with $a_1 \ge a_2 \ge \cdots \ge a_n$)
$i := 1$
while $i \le n$ and $a_i < x$
    $i := i + 1$
if $i = n + 1$ then first := 0
else if $a_i > x$ then first := 0
else
    first := $i$
    $i := i + 1$
    while $i \le n$ and $a_i = x$
        $i := i + 1$
    last := $i - 1$
{see above for the interpretation of the variables}
```

b) The number of comparisons depends on the data. Roughly speaking, in the worst case we have to go all the way through the list. This requires that x be compared with each of the elements, a total of n comparisons (not including bookkeeping). The situation is really a bit more complicated than this, but in any case the answer is $O(n)$.

6. **a)** We follow the instructions given. If n is odd then we start the loop at $i = 2$, and if n is even then we start the loop at $i = 3$. Within the loop, we compare the next two elements to see which is larger and which is smaller. The larger is possibly the new maximum, and the smaller is possibly the new minimum.

b)
```
procedure clever smallest and largest($a_1, a_2, \ldots, a_n$ : integers)
if $n$ is odd then
        $min := a_1$
        $max := a_1$
else if $a_1 < a_2$ then
        $min := a_1$
        $max := a_2$
else
        $min := a_2$
        $max := a_1$
if $n$ is odd then $i := 2$ else $i := 3$
while $i < n$
        if $a_i < a_{i+1}$ then
                $smaller := a_i$
                $bigger := a_{i+1}$
        else
                $smaller := a_{i+1}$
                $bigger := a_i$
        if $smaller < min$ then $min := smaller$
        if $bigger > max$ then $max := bigger$
        $i := i + 2$
{ $min$ is the smallest integer among the input, and $max$ is the largest}
```

c) If n is even, then pairs of elements are compared (first with second, third with fourth, and so on), which accounts for $n/2$ comparisons, and there are an additional $2((n/2) - 1) = n - 2$ comparisons to determine whether to update *min* and *max*. This gives a total of $(3n - 4)/2$ comparisons. If n is odd, then there are $(n - 1)/2$ pairs to compare and $2((n - 1)/2) = n - 1$ comparisons for the updates, for a total of $(3n - 3)/2$. Note that in either case, this total is $\lceil 3n/2 \rceil - 2$ (see Exercise 7).

8. The naive approach would be to keep track of the largest element found so far and the second largest element found so far. Each new element is compared against the largest, and if it is smaller also compared against the second largest, and the "best-so-far" values are updated if necessary. This would require about $2n$ comparisons in all. We can do it more efficiently by taking Exercise 6 as a hint. If n is odd, set l to be the first element in the list, and set s to be $-\infty$. If n is even, set l to be the larger of the first two elements and s to be the smaller. At each stage, l will be the largest element seen so far, and s the second largest. Now consider the remaining elements two by two. Compare them and set a to be the larger and b the smaller. Compare a with l. If $a > l$, then a will be the new largest element seen so far, and the second largest element will be either l or b; compare them to find out which. If $a < l$, then l is still the largest element, and we can compare a and s to determine the second largest. Thus it takes only three comparisons for every pair of elements, rather than the four needed with the naive approach. The counting of comparisons is exactly the same as in Exercise 6: $\lceil 3n/2 \rceil - 2$.

10. Following the hint, we first sort the list and call the resulting sorted list $a_1, a_2, \ldots, a_n$. To find the last occurrence of a closest pair, we initialize *diff* to ∞ and then for i from 1 to $n - 1$ compute $a_{i+1} - a_i$. If this value is less than *diff*, then we reset *diff* to be this value and set k to equal i. Upon completion of this loop, a_k and a_{k+1} are a closest pair of integers in the list. Clearly the time complexity is $O(n \log n)$, the time needed for the sorting, because the rest of the procedure takes time $O(n)$.

12. We start with the solution to Exercise 37 in Section 3.1 and modify it to alternately examine the list from the front and from the back. The variables *front* and *back* will show what portion of the list still needs work. (After the k^{th} pass from front to back, we know that the final k elements are in their correct positions, and after the k^{th} pass from back to front, we know that the first k elements are in their correct positions.) The outer **if** statement takes care of changing directions each pass.

```
procedure shakersort($a_1, \ldots, a_n$)
front := 1
back := $n$
still_interchanging := true
while front < back and still_interchanging
        if $n + back + front$ is odd then {process from front to back}
                still_interchanging := false
                for j := front to back − 1
                        if $a_j > a_{j+1}$ then
                                still_interchanging := true
                                interchange $a_j$ and $a_{j+1}$
                back := back − 1
        else {process from back to front}
                still_interchanging := false
                for j := back down to front + 1
                        if $a_{j-1} > a_j$ then
                                still_interchanging := true
                                interchange $a_{j-1}$ and $a_j$
                front := front + 1
{ $a_1, \ldots, a_n$ is in nondecreasing order}
```

14. Lists that are already in close to the correct order will have few items out of place. One pass through the shaker sort will then have a good chance of moving these items to their correct positions. If we are lucky, significantly fewer than $n-1$ passes through the list will be needed.

16. Since $8x^3 + 12x + 100\log x \le 8x^3 + 12x^3 + 100x^3 = 120x^3$ for all $x > 1$, the conclusion follows by definition.

18. This is a sum of n things, each of which is no larger than $2n^2$. Therefore the sum is $O(2n^3)$, or more simply, $O(n^3)$. This is the "best" possible answer.

20. Let us look at the ratio $n^n/n!$. We can write this as

$$\frac{n}{n} \cdot \frac{n}{n-1} \cdot \frac{n}{n-2} \cdots \frac{n}{2} \cdot \frac{n}{1}.$$

Each factor is greater than or equal to 1, and the last factor is n. Therefore the ratio is greater than or equal to n. In particular, it cannot be bounded above by a constant C. Therefore the defining condition for n^n being $O(n!)$ cannot be met.

22. By ignoring lower order terms, we see that the orders of these functions in simplest terms are 2^n, n^2, 4^n, $n!$, 3^n, and n^4, respectively. None of them is of the same order as any of the others.

24. We know that any power of a logarithmic functions grows more slowly than any power function (with power greater than 0), so such a value of n must exist. Begin by squaring both sides, to give $(\log n)^{2^{101}} < n$, and then because of the logarithm, let $n = 2^k$. This gives us $k^{2^{101}} < 2^k$. Taking logs of both sides gives $2^{101}\log k < k$. Letting $k = 2^m$ gives $2^{101} \cdot m < 2^m$. This is almost true when $m = 101$, but not quite; if we let $m = 108$, however, then the inequality is satisfied, because $2^7 > 108$. Thus our value of n is $2^{2^{108}}$, which is very big! Notice that there was not much wiggle room in our analysis, so something significantly smaller than this will not do.

26. The first five of these functions grow very rapidly, whereas the last four grow fairly slowly, so we can analyze each group separately. The value of n swamps the value of $\log n$ for large n, so among the last four, clearly $n^{3/2}$ is the fastest growing and $n^{4/3}(\log n)^2$ is next. The other two have a factor of n in common, so the issue is comparing $\log n \log\log n$ to $(\log n)^{3/2}$; because logs are much smaller than their argument, $\log\log n$ is much smaller than $\log n$, so the extra one-half power wins out. Therefore among these four, the desired order is $\log n \log\log n$, $(\log n)^{3/2}$, $n^{4/3}(\log n)^2$, $n^{3/2}$. We now turn to the large functions in the list and take the logarithm of each in order to make comparison easier: $100n$, n^2, $n!$, 2^n, and $(\log n)^2$. These are easily arranged in increasing big-O order, so our final answer is

$$\log n \log\log n, \quad (\log n)^{3/2}, \quad n^{4/3}(\log n)^2, \quad n^{3/2}, \quad n^{\log n}, \quad 2^{100n}, \quad 2^{n^2}, \quad 2^{2^n}, \quad 2^{n!}.$$

28. The greedy algorithm in this case will produce the base c expansion for the number of cents required (except that for amounts greater than or equal to c^{k+1}, the c^k coins must be used rather than nonexistent c^i coins for $i > k$). Since such expansions are unique if each digit (other than the digit in the c^k place) is less than c, the only other ways to make change would involve using c or more coins of a given denomination, and this would obviously not be minimal, since c coins of denomination c^i could be replaced by one coin of denomination c^{i+1}.

30. **a)** We follow the hint, first sorting the sequence into $a_1, a_2, \ldots, a_n$. We can then loop **for** $i := 1$ **to** $n-1$ and within that **for** $j := i+1$ **to** n and for each such pair (i,j) use binary search to determine whether $a_j - a_i$ is in the sorted sequence.

b) Recall that sorting can be done in $O(n \log n)$ time and that binary searching can be done in $O(\log n)$ time. Therefore the time inside the loops is $O(n^2 \log n)$, and the sorting adds nothing appreciable to this, so the efficiency is $O(n^2 \log n)$. This is better than the brute-force algorithm, which clearly takes time $\Omega(n^3)$.

32. We will prove this essentially by induction on the round in which the woman rejects the man under consideration. Suppose that the algorithm produces a matching that is not male optimal; in particular, suppose that Joe is not assigned the valid partner highest on his preference list. The way the algorithm works, Joe proposes first to his highest-ranked woman, say Rita. If she rejects him in the first round, it is because she prefers another man, say Sam, who has Rita as his first choice. This means that any matching in which Joe is married to Rita would not be stable, because Rita and Sam would each prefer each other to their spouses. Next suppose that Rita leaves Joe's proposal pending in the first round but rejects him in favor of Ken in the second round. The reason that Ken proposed to Rita in the second round is that he was rejected in the first round, which as we have seen means that there is no stable matching in which Ken is married to his first choice. If Joe and Rita were to be married, then Rita and Ken would form an unstable pair. Therefore again Rita is not a valid partner for Joe. We can continue with this argument through all the rounds and conclude that Joe in fact got his highest choice among valid partners: Anyone who rejected him would have been part of an unstable pair if she had married him.

It remains to prove that the deferred acceptance algorithm in which the men do the proposing is female pessimal, that each woman ends up with the valid partner ranking lowest on her preference list. Suppose that Jan is matched with Ken by the algorithm, but that Jan ranks Ken higher than she ranks Jerry. We must show that Jerry is not a valid partner. Suppose there were a stable matching in which Jan was married to Jerry. Because Ken got the highest ranked valid partner he could, in this hypothetical situation he would be married to someone he liked less than Jan. But then Jan and Ken would be an unstable pair. So no such matching exists.

34. This follows immediately from Exercise 32 because the roles of the sexes are reversed.

36. This exercise deals with a problem studied in the following paper: V. M. F. Dias, G. D. da Fonseca, C. M. H. de Figueiredo, and J. L. Szwarcfiter, “The stable marriage problem with restricted pairs,” *Theoretical Computer Science* **306** (2003), 391–405. See that article for details, which are too complex to present here.

38. Consider the situation in Exercise 37. We saw there that it is possible to achieve a maximum lateness of 5. If we schedule the jobs in order of increasing slackness, then Job 4 will be scheduled fourth and finish at time 65. This will give it a lateness of 10, which gives a maximum lateness worse than the previous schedule.

40. Clearly we cannot gain by leaving any idle time, so we may assume that the jobs are scheduled back-to-back. Furthermore, suppose that at some point in time, say t_0, we have a choice between scheduling Job A, with time $t_{\rm A}$ and deadline $d_{\rm A}$, and Job B, with time $t_{\rm B}$ and deadline $d_{\rm B}$, such that $d_{\rm A} > d_{\rm B}$, one after the other. We claim that there is no advantage in scheduling Job A first. Indeed, the lateness of any job other than A or B is independent of the order in which we schedule these two jobs. Suppose we schedule A first. Then its lateness, if any, is $t_0 + t_{\rm A} - d_{\rm A}$. This value is clearly exceeded by the lateness (if any) of B, which is $t_0 + t_{\rm A} + t_{\rm B} - d_{\rm B}$. This latter value is also greater than both $t_0 + t_{\rm B} - d_{\rm B}$ (which is the lateness, if any, of B if we schedule B first) and $t_0 + t_{\rm A} + t_{\rm B} - d_{\rm A}$ (which is the lateness, if any, of A if we schedule B first). Therefore the possible contribution toward maximum lateness is always worse if we schedule A first. It now follows that we can always get a better or equal schedule (in terms of minimizing maximum lateness) if we swap any two jobs that are out of order in terms of deadlines. Therefore we get the best schedule by scheduling the jobs in order of increasing deadlines.

42. We can assign Job 1 and Job 4 to Processor 1 (load 10), Job 2 and Job 3 to Processor 2 (load 9), and Job 5 to Processor 3 (load 8), for a makespan of 10. This is best possible, because to achieve a makespan of 9, all three processors would have to have a load of 9, and this clearly cannot be achieved with the given running times.

44. In the pseudocode below, we have reduced the finding of the smallest load at a certain point to one statement; in practice, of course, this can be done by looping through all p processors and finding the one with smallest L_j (the current load). The input is as specified in the preamble.

```
**procedure** *assign*$(p, t_1, t_2, \ldots, t_n)$
**for** $j := 1$ **to** $p$
    $L_j := 0$
**for** $i := 1$ **to** $n$
    $m :=$ the value of $j$ that minimizes $L_j$
    assign job $i$ to processor $m$
    $L_m := L_m + t_i$
```

46. From Exercise 43 we know that the minimum makespan L satisfies two conditions: $L \geq \max_j t_j$ and $L \geq \frac{1}{p}\sum_{j=1}^{n} t_j$. Suppose processor i^* is the one that ends up with the maximum load using this greedy algorithm, and suppose job j^* is the last job to be assigned to processor i^*, giving it a total load of T_{i^*}. We must show that $T_{i^*} \leq 2L$. Now at the point at which job j^* was assigned to processor i^*, its load was $T_{i^*} - t_{j^*}$, and this was the smallest load at that time, meaning that every processor at that time had load at least $T_{i^*} - t_{j^*}$. Adding up the loads on all p processors we get $\sum_{i=1}^{p} T_i \geq p(T_{i^*} - t_{j^*})$, where T_i is the load on processor i at that time. This is equivalent to $T_{i^*} - t_{j^*} \leq \frac{1}{p}\sum_{i=1}^{p} T_i$. But $\sum_{i=1}^{p} T_i$ is the total load at that time, which is just the sum of the times of all the jobs considered so far, so it is less than or equal to $\sum_{j=1}^{n} t_j$. Combining this with the second inequality in the first sentence of this solution gives $T_{i^*} - t_{j^*} \leq L$. It remains to figure in the contribution of job j^* to the load of processor i^*. By the first inequality in the first sentence of this solution, $t_{j^*} \leq L$. Adding these two inequalities gives us $T_{i^*} \leq 2L$, as desired.